Blues Bot: CS182 Final Project Writeup

Bennett Parsons, Jeremy Welborn

December 8, 2017

For source code, visit `https://github.com/bennettparsons/jazz-bot`

1 Introduction

Music, specifically music composition, has become an exciting area of research in the artificial intelligence community. The biggest players have started to dig into this domain rife for AI, like Google Brain in its widely publicized Magenta project. We were interested in this area, specifically improvisation or improvised composition in the jazz idiom. This is perhaps a non-standard problem, because the notion of a "solution", of "correctness" is nebulous. In jazz, a player does not have a completely pre-computed plan or policy before their turn comes to solo. While we acknowledge a tightly-scoped agent to create jazz will have to be reductive in some sense, we aimed to implement an agent that could, given chord changes, create a convincing solo.

We went through several iterations of thought on how to implement such an agent. We first considered an agent that would be trained using deep learning techniques like recurrent neural networks. This is discussed in additional detail in the background section. Training requires data and the requisite data (single-voiced transcriptions of jazz solos in a convenient file format like MIDI) were not readily available. We then considered an agent that would not be trained, but rather would search some space of candidate solos using heuristics, functions that would reward things like tonal and rhythmic ideas reflective of idiomatic jazz. The idea of a reward is inherent in reinforcement learning (RL), as well as in local search. After considering which musical representations made sense, we stuck with well-known variations on local search, including simulated annealing and genetic algorithms — algorithms discussed in depth in CS182.

Our agent relies on an intuitive notion of state, which is a complete "solution" or solo over a single measure (which consists of exactly one chord). We were able to transform a completely random sequence of notes into something coherent and cohesive and we were able to string together solutions in a compelling way — crudely modeling the way in which a human might improvise through a jazz form.

Ultimately, while our agent's outputs are not easily "scorable", they were extremely exciting for us to hear, tweak, and hear again. While our scope was in some sense "restricted to" a few jazz forms where our heuristics make the most sense, it is easily extendable to other genres. As discussed in the last section, the blues bot could prove to be a particularly useful compositional and pedagogical tool.

2 Background and Related Work

A survey of approaches can be found here, in which the author cites a couple of techniques in machine learning for musical improvisation. [2]

Among these, Markov models have been extensively explored. Early mathematical music theorist Iannis Xenakis was responsible for trailblazing work in this area. [5] However, neural networks, especially recurrent neural networks, have found themselves at the core of modern musical AI in the literature. Early efforts here found that unlike with Markov models, neural nets could easily extrapolate on sub-solos found in the training data. [3] Improvements on this approach relied on subclasses of RNNs like long short-term memory RNNs to propagate ideas throughout an improvised piece. [1] Since then, an extensive amount of research has explored neural networks applications in music.

Still, like in our approach as we discuss in detail below, some work has been done in local search approaches for improvisation. These efforts rely on different representations, but are useful for inspiration. [6, 4]

3 Problem Specification

As we are in the domain of music composition, our specification of the problem was quite broad: can we generate musical output that is a *convincing* representation of jazz improvisation? We restrict the scope of the problem to simply improvising over the blues, a 12 bar chord progression widely accepted as a simple and representative jazz form.

Given more time, we could imagine adapting the "Turing Test" to fit this problem. We could proceed with the following experiment to accept or reject our algorithm as a convincing representation of jazz. Gather a number of blues solo transcriptions of accepted jazz greats and match them side by side with outputs of the computer program. Play them both, back to back, to a human subject. Both outputs should have the same sound format (ideally MIDI) so that we only compare use of melody and rhythm (performance itself is an entirely different problem). The human is asked to decide which solo was played by the human. After conducting this experiment for a sufficient number of different human subjects and musical material, we accept the machine if humans could not reliably distinguish the solo performed by a jazz great and that by our machine.

Though we were unable to formally conduct this test, we were able to generate some convincing outputs that may very well have performed respectably on such a Turing Test.

4 Approach

4.1 Choosing Our Algorithm

As discussed in the introduction, we decided to implement local search algorithms, rather than Markov models and RL, to construct our data-independent, heuristic oriented agent. While an RL agent would have the benefit of learning a policy that could conceivable improvise in real time, the cost of representing the problem as an MDP was simply too unintuitive. For example, if a state was simply a note and a chord, then there could be no notion of value (since notes are not inherently good or bad by themselves). If a state was a sequence of notes, then the state space would be far too large. Thus, a local search representation — building up the solo by finding solutions to subproblems, measure by measure — was most practical.

In order to direct our local search toward a specific type of solution (i.e. jazz music), we designed a set of heuristics that could act on a solution and rate how well it performed along a particular musical metric. We will refer to these as feature evaluation functions throughout this paper. Our evaluation function for a single "solution" x is $\sum_i f_i(x)$ for feature evaluation functions f_i.

4.2 Feature Evaluation Functions

Though we initially imagined creating more complex feature evaluation functions, the two described below performed remarkably well, and appeared to capture enough musical elements to produce respectable jazz improvisations.

- *Tonality:* how well does the solo the harmonic vocabulary of the current chord? These ratings will be determined largely from the accepted jazz theory of how to solo over particularly kinds of chords. The hierarchy of relative rewards is roughly as follows:

  1. 3rds and 7ths of chord
  2. tension (provided it resolves correctly)
  3. any chord tone
  4. any note within the scale
  5. any note not in the above

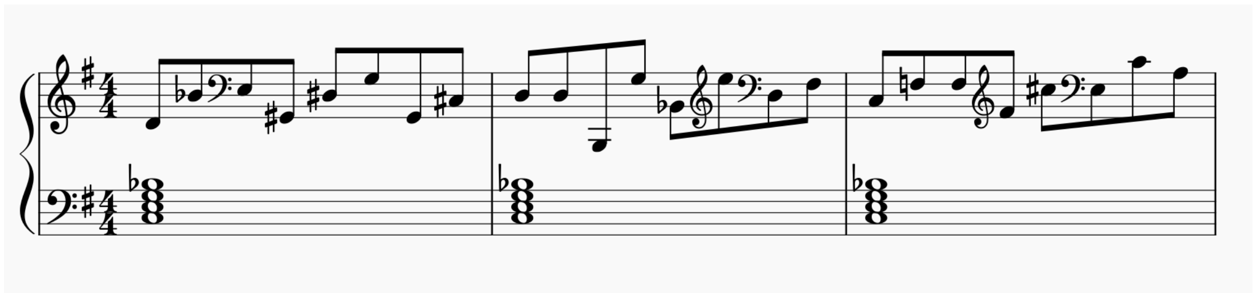
  Note that the tonality evaluator works per note, assigning a relative score to each note of the solution and returning the sum.

- *Contour:* how well does the solo capture the concept of a musical line? There should not be very many large intervallic leaps, and there should be an interesting mix of rising and falling lines. Our implementation does the following, with no particular hierarchy:

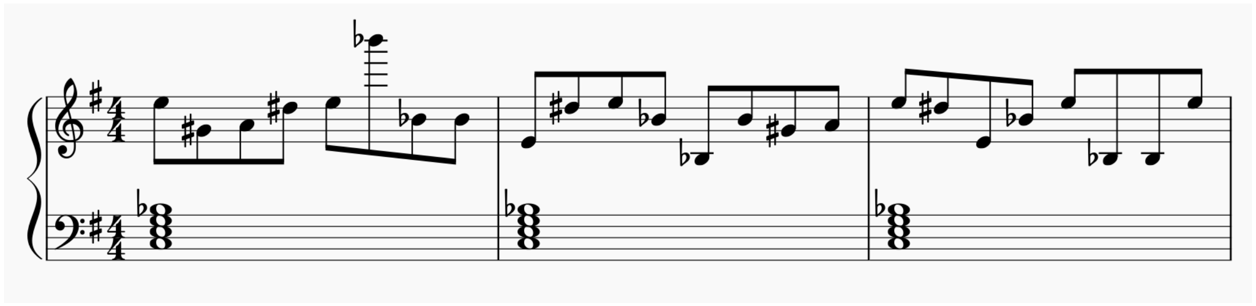
  1. incentivize varied contour (rising and falling notes) and intervals
  2. slightly incentivize small intervals over large ones
  3. slight penalization for repeating notes
  4. disincentivize leaps larger than an octave
  5. incentivize a general downward or upward line

  Unlike tonality, the contour evaluator acts on the entire solution at once, applying the above scoring heuristics to sets of notes within the solution.

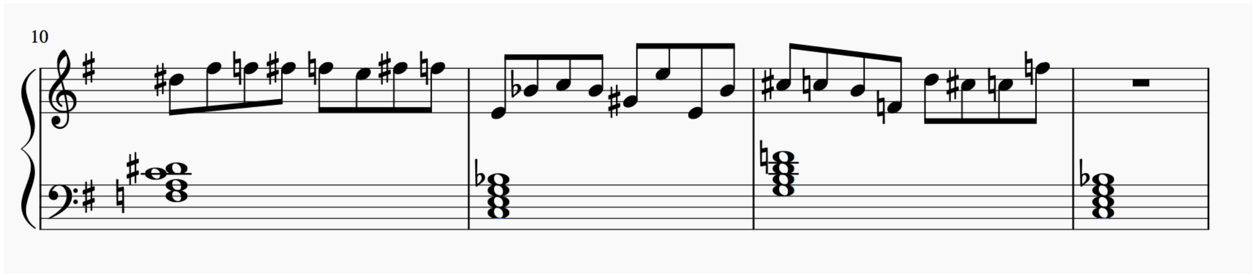
For more details on implementation, see `search.py`. For the effect of these heuristic functions — within the context of local search — see Figure 1. Notice how the combination of the tonal and the contour evaluators drives the random output toward something that resembles an accepted jazz bebop line.



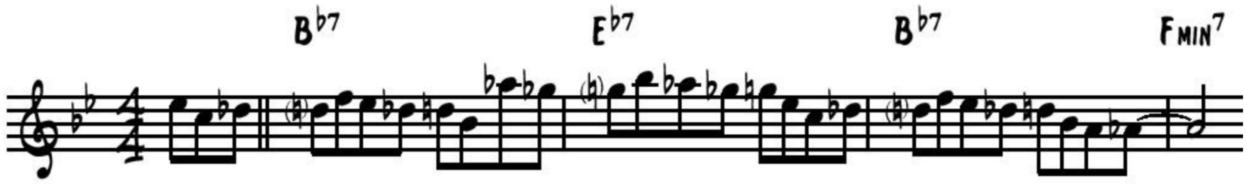
(a) Unfiltered stream of 1/8 notes



(b) Filtering by tonal evaluation



(c) Filtering by tonal and contour evaluation



(d) Example jazz improvisation

Figure 1: Effect of Feature Evaluation Functions on Output - notice how the feature evaluation functions select for solutions resembling the sample (1d) in both shape and harmonic vocabulary

4.3 Local Search Implementations

We decided to implement two different local search algorithms: simulated annealing and a genetic algorithm. Our implementations were standard, following intuitively from our problem representation (see pseudocode below). Our first approach defined a solution as a sequence of eight pitches (each with a duration of 1/8, so as to fill an entire measure). As you will see in the subsequent section, we soon changed our representation to handle solutions of any size between 1 and 8.

Algorithm 1 Simulated Annealing

```
procedure SA($init\_sol$)
    $curr\_sol \leftarrow init\_sol$
    $best\_sol \leftarrow curr\_sol$
    for $i$ : 1..200 do
        $candidate\_sol \leftarrow neighbor(curr\_sol)$
        if $value(candidate\_sol) > value(curr\_sol)$ then
            $curr\_sol \leftarrow candidate\_sol$
        else
            if $cooling(i)$ then
                $curr\_sol \leftarrow candidate\_sol$
            end if
        end if
        if $value(curr\_sol) > value(best\_sol)$ then
            $best\_sol \leftarrow curr\_sol$
        end if
    end for
    return $best\_sol$
end procedure
procedure NEIGHBOR($sol$)
    $i \leftarrow sample(len(sol))$
    $sol[i] = gaussian(sol[i], 3)$
    return $sol$
end procedure
```

5 Experiments

Due to the subjective nature of our goal, we could only evaluate our success in terms of our own musical interpretations of the outputs. Furthermore, there was no absolute scale for success. Thus, in the same way that a musician might practice to improve a particular aspect of his playing, we worked in cycles of evaluation and feature development. This involved analyzing our output, isolating facets that were musically weak, unintuitive or unbecoming of the jazz idiom, and adding features to our bot to improve these aspects. In this section, we discuss the shortcomings our bot had along the way and the step we took address these issues.

Algorithm 2 Genetic Algorithm

```
procedure GA($init\_sol$)
    $generations \leftarrow 10$
    $generation\_sz \leftarrow 10$
    $population \leftarrow generate\_population(init\_sol)$
    for $generations$ do
        $successor\_pop$
        for $generation\_sz$ do
            $mom, dad \leftarrow sample\_from\_population\_by\_fitness$
            $child \leftarrow \text{reproduce}(mom, dad)$
            $child \leftarrow \text{mutate}(child)$
            $successor\_pop \leftarrow successor\_pop \cup child$
        end for
        $population \leftarrow successor\_pop$
    end for
    return $\arg\max_{sol \in population} value(sol)$
end procedure
procedure REPRODUCE($mom, dad$)
    $i \leftarrow sample(len(mom))$
    return $mom[: i] + dad[i :]$
end procedure
procedure MUTATE($child$)
    $i \leftarrow sample(len(child))$
    $child[i] = gaussian(sol[i], 3)$
    return $child$
end procedure
```

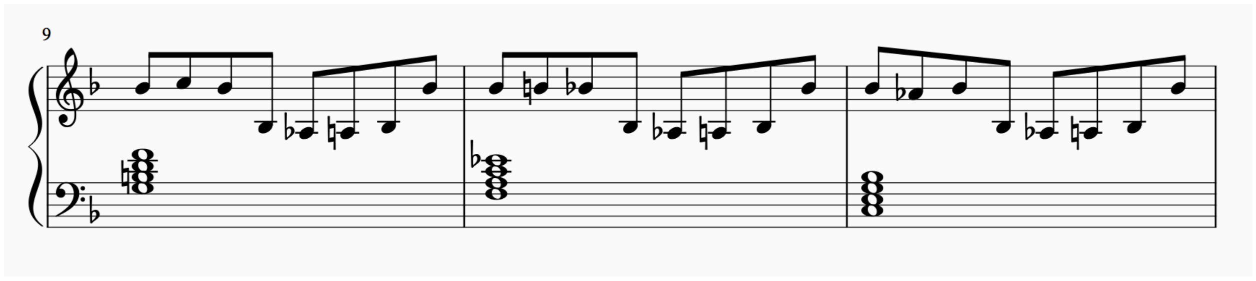


Figure 2: Initialize with Previous Solution

5.1 Initialization

Our first observation was that the solo lacked coherency from one measure to the next. This made sense since our initial approach simply pasted together unrelated solutions to subproblems — not at all capturing any sort of sensitivity to what has been played previously. We needed to make solutions to supbroblems somehow dependent on the past.

Our solution to this problem was simple, yet remarkably effective. Rather than run local search on our subproblems independently, in parallel, we decided to run them iteratively, building up the solo one measure at a time. This allowed us to initialize each subproblem with the previous subproblem's solution. If we kept the number of iterations of local search small enough — 200 seemed to work well for simulated annealing, and 10 generations of size 10 worked well for the genetic algorithm — then there was a built-in dependence between subproblems. See Figure 2 for an example of a coherent and compelling phrase produced using this technique.

5.2 Resolutions

Another nature issue with our initial naive solution was that our contour function was being applied discontinuously. Since this feature evaluation function analyzes relationships between pairs of notes, and since it is only applied to notes within a single measure, the relationship between the last note of a measure the first of the next was not being evaluated.

We addressed this by introducing a modified version of the contour function, which, given a solution, would select a resolution pitch sampled from a weighted distribution of pitches it would be likely to resolve to.

Once a resolution pitch was selected, we initialized the subsequent solution with this pitch as its first, and modified our search algorithms to be able to handle "fixed notes," which could not change during the local search process. This produced output as seen in Figure 3 where line continuity between measures is well preserved.

5.3 Rhythm

As mentioned previously, our initial approach neglected rhythm as a musical parameter by restricting the "size" of solutions to be 8 notes (thereby causing all notes to be 1/8 notes within the 4/4 time signature we were using).

We again took the simplest solution possible — which, once again, produced surprisingly effective results! Rather than fix the "size" of a solution to 8 notes, we allowed each subproblem to

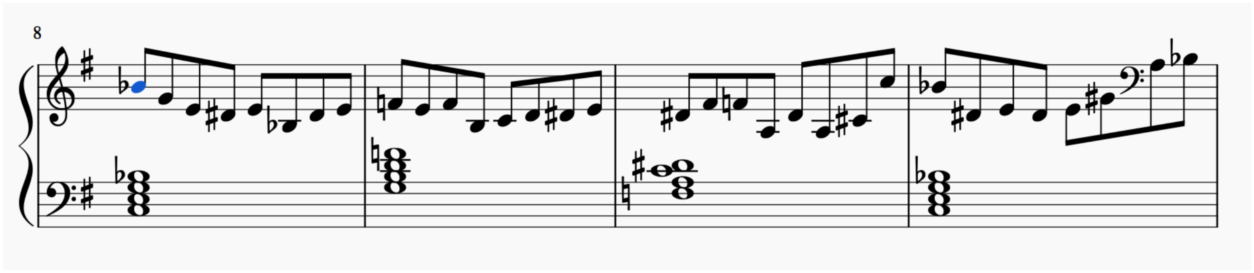


Figure 3: Resolutions Across Measures

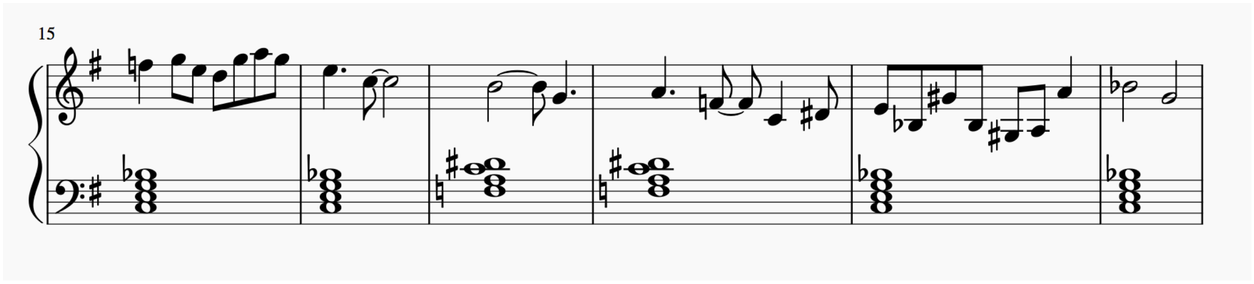


Figure 4: New Rhythms

take on a size selected randomly from 1-8. To support this, we first modified the feature evaluation functions to take in sets of pre-determined parameters that depend solution size (these were hard-coded in `theory.py`). We then ran the same local search algorithms, but introduced the following post-processing procedure which made the solution viable (i.e. made the collection of notes last for the duration of one measure):

Algorithm 3 Rhythm Implementation

```
procedure RHYTHM($sol$)
    while $duration(sol)$ not equal 4 do
        $i \leftarrow sample(len(sol))$
        if $duration(sol) < 4$ then
            $add\_duration(sol[i])$
        else
            $subtract\_duration(sol[i])$
        end if
    end while
    return $sol$
end procedure
```

Thus, rhythms were generated completely randomly. Yet since this implementation was compatible with the previous notion of initialization, similar rhythms could propagate from measure to measure throughout the solo, giving it rhythmic, in addition to melodic, coherency. Figure 4 shows a brief snippet of such output.

5.4 Build

At this point, we were fairly pleased with our output on the standard 12 bar blues and wanted to see if we could add a notion of solo build across multiple choruses (repetitions of the chord progression). This was an important feature to implement because it gives the solo shape — just as the contour heuristic does, but on a larger scale.

Leveraging our implementation of rhythm led to a quite natural solution. Since solo build generally manifests as denser playing, we changed the way we initialized the size of subproblems. Specifically, we selected the size from a guassian distribution (with support [1,8]) whose mean was a function of the chorus. We used a simple linear mapping where: $\mu = \frac{current\_chorus}{total\_choruses} * 8$. We also set the standard deviation to decrease linearly (as a function of the current chorus). This would cause the solo to start off sparsely, but with much variation, and then gradually converge to a denser, more climatic texture toward the end of the solo. This simple change breathed life into the large form of the solo.

5.5 Simulated Annealing vs. Genetic Algorithm

"For algorithm-comparison projects: a section reporting empirical comparison results preferably presented graphically."

Ultimately, the biggest improvements in our bot were in subsequent iterations of representation and feature functions. However, we were interested in two separate classes of algorithms for local search: simulated annealing and genetic algorithms. Simulated annealing is a hill-climbing algorithm, i.e. it steps from neighbor to higher-valued neighbors, with a Metropolis variation whose intensity is time-dependent. That is, simulated annealing accepts transitions from a neighbor to a lower-valued neighbor with some small probability that becomes smaller through time in an attempt to avoid the trap of local optima that is likely to occur early on in search. Genetic algorithms are similar to beam searches (or rather stochastic beam searches) in that they track several search threads, but advance to subsequent states by "crossing over" highly scoring solutions in an attempt to extract different features of compelling sub-solutions. While a simulated annealing approach is standard and worked well as an out-of-the-box search algorithm, we were interested in the idea that a genetic algorithm could combine interesting licks, especially interesting licks with intervallic jumps that we would not otherwise encounter using the standard search approach.

However, it is worth noting that there are a large number of hyperparameters at play in our agent. Significantly, the number of iterations (or the number of "generations" in a genetic algorithm) would have a heavy influence on how similar measures were to previous measures, which is often desirable for the sake of solo continuity or coherence, as well as on how well a solution could incorporate new, interesting musical ideas. This tradeoff was not easily tunable.

We wrote a short script to compare how our algorithms score (i.e. score per note). Using the code's current hardcoded hyperparameters, we see that simulated annealing tends to score higher:

	Avg score per note in a solo
SA	4.33784054402
GA	3.78672686867

This is not so surprising. Since the genetic algorithm was much more computationally expensive, we ran far fewer iterations so measures tended to deviate less from previous measures in the genetic algorithm. Of course, we did not conduct a thorough hypothesis test, but the results of the script confirm our suspicions about this subjective tradeoff.

Indeed, as Russell and Norvig in AIMA alluded to, genetic algorithms are often cited for their crossover operation. We could not find certain evidence that crossing over was creating sequences of separate licks that could not have otherwise co-occurred. There is too much randomness to measure this, though we tried to investigate specific places where this could have been the case. In the end, there was no significant difference between interesting leaps that may have been the result of cross over in the genetic algorithm, with interesting leaps that appeared to evolve naturally in simulated annealing. This is likely a result of the flexibility of our contour evaluator, which only slightly disincentivized large leaps.

We also observed that in transitioning from neighbor to neighbor, our simulated annealing implementation tended to inject "more" randomness than the "mutation" operation in the genetic algorithm. For this reason too it seems we can confirm that the genetic algorithm tended to be a little less "free" in an improvisational sense.

Still, the character of these separate classes of algorithms did result in some distinct solo qualities and we are extremely interested in the extent to which tuning can help discern between the two.

6 Discussion

In this project, we implemented a data-independent jazz improviser, which used handcrafted, feature evaluation functions to drive randomly generated musical output toward a particular genre. To generate an entire solo, we discretized the musical form into a series of subproblems — single measures consisting of a single chord — which we could solve individually, and then paste together to form one solo. We implemented local search algorithms that explored random solutions to these subproblems, returning the solution that most resembled jazz, according to our heuristics. To craft more compelling solos, we added a number of simple features to this framework which had a profound effect on the output. These included initializing a measure's solution with the previous measure's result, and bridging the gap between measures by adding a resolution heuristic in which one subproblem would determine the best starting pitch for the subsequent. After implementing a simple mechanism for rhythm, we added a feature for solo build, which caused rhythms to become denser as the solo progressed, leading to a more realistic sense of long term shape and direction.

Considering the simple nature of this approach, our results were quite good. Though it is hard to give a convincing argument for this claim, those who heard our bot's output, and had a sense of the genre we were emulating, were quite convinced by results, and indeed startled that a bot without any training could produce substantive, convincing solos. This is a testament to a strong moral of software development that this project has shown us firsthand: the best solution is often the simplest.

That said, there are still a number of features we would have loved to have added, given more time. Among them would be a heuristic for motivic development. While we got some of this for free by initializing local search intelligently, good musical ideas would often be transformed beyond recognition within a few measures. One could imagine adding a heuristic that backtracks through all previous solutions, and rewards repetition of material previously played. Even simpler, we could have implemented more complex rhythms, using additional heuristics for selecting varied and/or syncopated rhythms.

While we chose to represent the musical style of jazz — indeed, a specific subset of jazz that most resembles a blend of swing and bebop — our methods could be extended to create a bot that improvises in any musical genre. Indeed, the only source file in our code base that describes our simple representation of jazz is `theory.py`, which hard-codes a number of parameters used in our feature evaluation functions. If we were implementing an improviser for rock, fusion, R/B, or any other kind of music, this would be the only file we would have to alter.

Historically, the problem of jazz improvisation has been approached using deep learning algorithms. While these approaches have enjoyed much success in recent years, this task has rarely been approached from a training-free perspective. A more widespread success of these kinds of approaches would have very interesting consequences. One could imagine pedagogical applications of this software, where a student learning jazz improvisation could emulate output from the bot that was produced by using only a specific set of heuristic functions. It is often difficult for students to learn directly from the greats because there is so much content in those solos, and it is difficult to isolate exactly how different techniques are utilized.

While deep learning approaches may ultimately prove most effective at imitating past styles, heuristic based approaches to music composition in general could be used to aid modern composers in their creation of new works of art, and even entirely new idioms of composition. In a world increasingly driven by large computing systems, which many people feel are replacing human ingenuity, it would be a refreshing turn of events if composers could work together with AI agents, creating new styles of music and pushing the boundaries of what we may consider great art, together.

Appendix

System Description

Abbreviated information / instructions: The code is written for Python 2.7. It has only one dependency that is not in the standard distribution, `midiutil` which is available for `pip` or `conda`. To play the MIDI piece that the code writes, you can upload the `.mid` file to an MIDI player on the web or quickly download MuseScore to see the score itself.

Reading the code:

Our agent speaks MIDI. It accepts a sequence of chords i.e. a chord progression and it returns a file in the MIDI format, the de facto standard for digitizing music. Our implementation relies on 5 Python files for the agent to run:

- `util.py` which contains helper functions for manipulating musical objects and for interfacing with the `midiutil` library
- `theory.py` which contains definitions and dictionaries for the theory concepts the code relies on as well as hard-coded values for our feature evaluation functions
- `structures.py` which contains classes for a `Note` and a `Chord` that are amenable for MIDI
- `problems.py` which contains a class for our proposed problem representation, i.e. a sequence of subproblems which are measures of one and only one chord
- `search.py` which contains implementation of our search algorithms as well as the heuristics they rely on

Running the code:

We've written the code to be run with Python 2.7, per the class requirement for assignments. In addition, only one non-standard library package is needed, `midiutil`. This is readily available with the python package manager `pip`. In addition, if you wish to hear the MIDI that the agent outputs, you can upload the MIDI to MIDI sequencers on the web or you can download MuseScore, free scorewriting software which we have used and is available here.

There is an additional script that contains a convenient command-line interface, `blues-bot.py`. Running `./blues-bot.py -h` (or `python blues-bot.py -h` in case your interpreter cannot be referenced from `/usr/bin/local`) writes to stdout:

```
usage: blues-bot.py [-h] [-a {SA,GA}] [-k {A,Bb,B,C,Db,D,Eb,E,F,Gb,G,Ab}]
                    [-p PROGRESSION [PROGRESSION ...]] [-n NUMBER_CHORUSES]
                    [-f FILENAME]

Run the blues bot!

optional arguments:
  -h, --help            show this help message and exit

  -a {SA,GA}, --algorithm {SA,GA}
```

```
                        run with simulated annealing (SA) or a genetic
                        algorithm (GA)

  -k {A,Bb,B,C,Db,D,Eb,E,F,Gb,G,Ab}, --key {A,Bb,B,C,Db,D,Eb,E,F,Gb,G,Ab}
                        specify the key, like C or Db

  -p PROGRESSION [PROGRESSION ...], --progression PROGRESSION [PROGRESSION ...]
                        specify a sequence of numerals and qualities for the
                        progression, separated by spaces

  -n NUMBER_CHORUSES, --number_choruses NUMBER_CHORUSES
                        specify the number of choruses to compose

  -f FILENAME, --filename FILENAME
                        specify the midi file to write to
```

Running `./blues-bot.py` with no flags defaults to a single chorus of a 12-bar blues in the key of C. If you do not wish to run the agent on your own progression, it is instructive to compare the outputs of the two algorithms implemented, simulated annealing and a genetic algorithm. To do this, run `./blues-bot.py -a SA` or `./blues-bot.py -a GA` respectively. This will write a solo with the default title `blues-bot-solo.mid` to your current directory. This piece can be played in MuseScore or online as described above.

Group Makeup

The project participants were Bennett Parsons and Jeremy Welborn. While much of the programming was done in pair while working together, Bennett was chiefly responsible for implementing the music theoretic heuristic functions where he has extensive experience and Jeremy was responsible for implementing the search tools that the agent relies on.

References

[1] Douglas Eck and Jürgen Schmidhuber. Finding temporal structure in music: Blues improvisation with lstm recurrent networks.

[2] Robert Keller. Machine learning applied to musical improvisation. 2013.

[3] Peter Todd. A connectionist approach to algorithmic composition. *Computer Music Journal*, 13(4):27–43, 1989.

[4] Gil Weinberg, Mark Godfrey, Alex Rae, and John Rhoads. A real-time genetic algorithm in human-robot musical improvisation. *Computer Music Modeling and Retrieval*, pages 351–359, 2007.

[5] Iannis Xenakis. Formalized music: Thought and mathematics in composition. 1992.

[6] Ender zcan and Trker Eral. A genetic algorithm for generating improvised music. *Artificial Evolution*, pages 266–277, 2007.

For reference, each of these citations can be found online: [1], [2], [3], [4], [5], [6].